A Low-Overhead Framework for Inexpensive Embedded Control Systems

Ivan Cibrario Bertolotti*, Tingting Hu§, and Gilda Ghafour Zadeh Kashani*

* National Research Council of Italy – IEIIT, Torino, Italy
Email: {ivan.cibrario, gilda.kashani}@ieiit.cnr.it
§ University of Luxembourg – Faculty of Science, Technology and Communication, Luxembourg
Email: tingting.hu@uni.lu

***Abstract*—Embedded control systems are becoming more and more popular, especially in relatively inexpensive consumer products, like home appliances and building automation controllers. As a consequence, there is an ever increasing desire to reduce firmware development time and cost, without hampering reliability and performance. In this paper, a low-overhead firmware development framework is proposed, which allows programmers to develop and deploy typical real-time control software faster than using plain C-language programming. At the same time, experimental results confirm the framework's efficiency and applicability even to low-end microcontrollers.**

Keywords–Embedded control systems; Logic controllers; Firmware development frameworks.

I. Introduction and Related Work

The ever-increasing popularity of inexpensive control systems—mainly driven by consumer products, which nowadays invariably include at least one embedded processor—brings new challenges to firmware development. This market segment is characterized by fierce competition that forces vendors to reduce costs and shrink time to market, a fact that strongly encourages the introduction of more abstraction in the development process.

This often entails the adoption of new programming techniques. For instance, [1] discusses a framework that works according to an event-driven paradigm and UML statecharts. Similarly, programming environments for Programmable Logic Controllers (PLCs) [2], either proprietary [3] or open-source [4], are indeed quite powerful. Nevertheless, they force programmers to learn dedicated languages they are unlikely familiar with, for instance, the ones defined in [5]. Moreover, the cost of a PLC is typically much higher than the cost of a microcontroller-based board of equivalent processing power. In the case of [6], the framework indeed adopts a simplified dialect of the C++ language. However, the programming model consists of a single main-loop that contains the code to be executed, and must be suitably extended to support real-time multitasking [7].

Staying with the easiest and probably most widespread embedded systems programming language—that is, the C language [8]—brings the additional advantage of making readily available a variety of sophisticated open-source firmware components, ranging from real-time operating systems (RTOS) [9] to filesystems [10] and TCP/IP protocol stacks [11]. Previous proposals, like the one described in [12], succeeded to leverage these components and speed up firmware development, but they still lack much needed abstraction. In the paper, a firmware development framework is proposed, whose goal is to shield application-level firmware developers from most low-level architectural details and abstract from Input–Output (I/O) mechanisms through a flexible configuration system. At the same time, programmers may still use the C language, thus reaching a convenient trade-off between the two goals discussed previously.

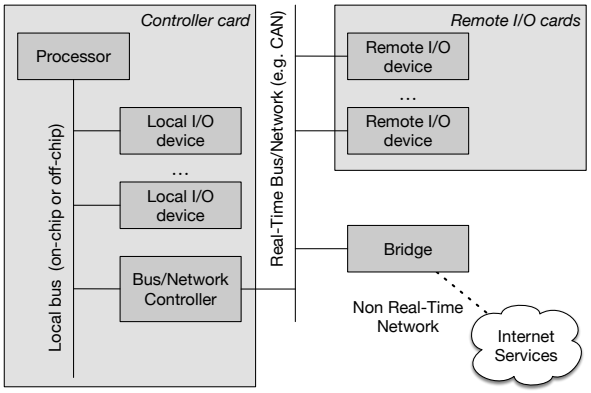

Figure 1. Hardware architecture of a typical distributed control system.

Real-time execution models have been the subject of considerable debate since a long time [13]. Even though other, more sophisticated models have been proposed [14], the framework proposed in this paper sticks with a traditional *cyclic executive* [15] for the real-time control part to boost performance and reduce overheads. In order to overcome the inflexibility of the method when considered alone [13] [14], it has been blended with a more general task-based system, coordinated by the underlying FreeRTOS RTOS. The paper is organized as follows. Section II describes how the framework has been designed and discusses its architecture. Next, Section III provides more information about its implementation, focusing on two critical aspects. Section IV presents the experimental evaluation, as well as the related measurement method, and Section V concludes the paper.

II. Framework Design and Architecture

A. Main Features and Design Guidelines

In recent years, a generalized trend in the design and implementation of embedded control systems has been to shift from a concentrated to a distributed I/O paradigm. Accordingly, as shown in Figure 1, the framework supports *local* I/O devices, reachable by the processor by means of an on-chip or off-chip

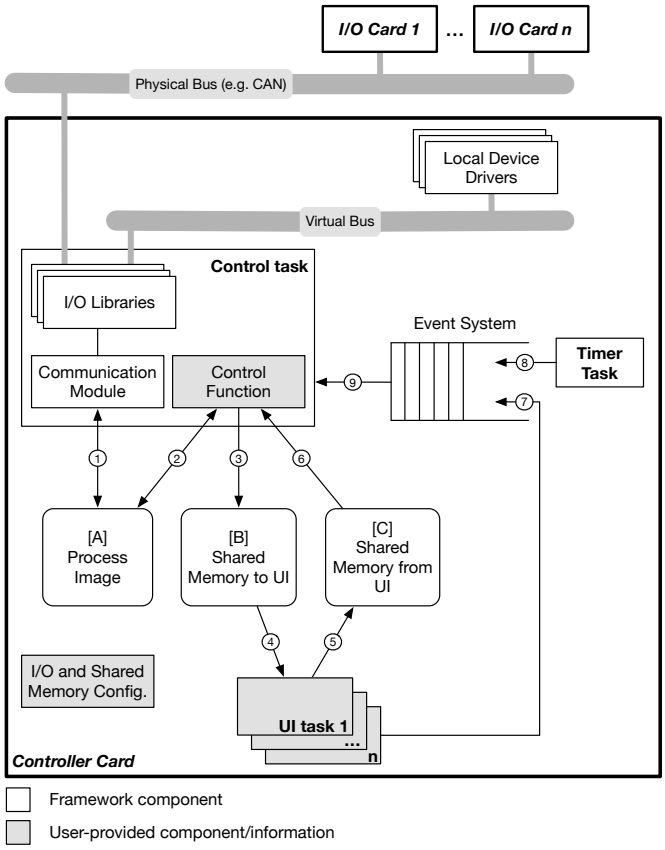

Figure 2. Simplified framework architecture.

bus local to the controller card, as well as *remote* I/O cards and devices, which are connected to the processor by means of a real-time communication network.

Furthermore, the framework enables firmware programmers to focus on the application-level part of design and development, such as the control algorithm. More specifically, it allows firmware programmers to operate on abstract *process image* (PI) variables, which represent the state of system inputs and outputs, regardless of how and where they are physically implemented. Last, but not least, even though some features of the framework resemble the ones provided by PLCs [2], in our case programmers shall be able to develop embedded control code using the C language [8] directly. Therefore, the above-mentioned PI variables are made available to programmers as any other, ordinary C-language variable is.

An additional feature of the framework, shown in the bottom-right part of Figure 1, is to provide access to the Internet through a bridge card. This feature is extremely important to implement key functionality, such as data logging, supervision, and firmware upgrades. However, it will not be further discussed in this paper for conciseness.

B. Framework Architecture

As it can be seen in Figure 2, the framework coordinates two groups of tasks that together form the complete control system and its user interface (UI). The first group of task consists of a *control* task, as well as a *timer* task. These two tasks implement the real-time control cycle, to be described in more details in Secton II-C. The only user-written code in this group is an embedded *control function*, which executes the control algorithm cyclically when invoked by the framework. A memory area, denoted as [A] in the figure, holds the aforementioned PI variables and is accessible to control task components only. The second group of tasks is outside the scope of the framework. All tasks in this group are completely user-written—except for what concerns their interface to the first group—and they cooperate to realize the system UI. Communication and data sharing between the two groups take place by means of two shared memory areas, denoted as [B] and [C] in Figure 2, one for each transfer direction (to or from the UI tasks, respectively).

The same figure also depicts the main data flows internal to the control system and managed by the framework, identified by means of circled numbers. More specifically:

- The *communication module* and the control function, both within the control task, have read-write access to memory [A] through data flows ① and ②, respectively. The role of the control function has already been mentioned before. On the other hand, the communication module is responsible of updating PI variables cyclically. In order to transparently support multiple physical implementations of those variables, all I/O functions are mediated by appropriate I/O libraries. For instance, the current implementation supports Modbus CAN [16] I/O cards, as well as local I/O devices by means of device drivers.
- Data flows ③ through ⑥ implement communication with the UI tasks. Unlike memory [A], memories [B] and [C] are protected against concurrent access by means of mutual exclusion locks with definite time-outs on the real-time side. The decision of having two memory areas, each supporting a unidirectional data flow, is useful to reduce lock contention and improve granularity. At the same time, keeping memory [A] totally separate from [B] and [C] also works as a safeguard against unintentional or unauthorized access to PI variables by non real-time tasks. Data sharing among these areas is implemented in a controlled way by the framework, through the control task itself, by periodically mirroring a pre-configured subset of PI variables to/from [B] and [C].
- As it will be better discussed in Section II-C, the control task implements a traditional control cycle. On the other hand, UI tasks are event driven and, sporadically, may need to convey expedited information to the control task itself. This is done by generating and sending *sporadic events* to it, according to flow ⑦ of Figure 2. As a consequence, the control task has to react to two event sources: one *cyclic*, and one *sporadic*. Cyclic events are generated by the timer task and correspond to flow ⑧ in the figure. These two event sources are prioritized and combined into flow ⑨ by a dedicated *event system*, which also takes into account some peculiarities and shortcomings of the underlying RTOS. More details about event system implementation are given in Section III-B.

C. Control Cycle

Figure 3 contains a more detailed view of the control cycle implemented by the framework, along with sporadic events handling. Individual control cycles are triggered by

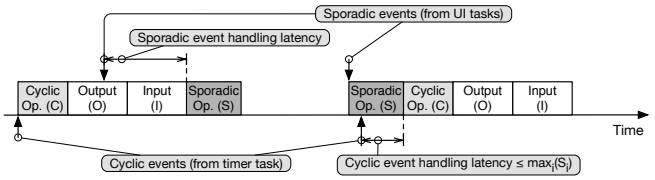

Figure 3. Control cycle and sporadic event handling.

```
BOARD_INSTANCE_LOCAL(
  sample_class, instance_name, 0,
  LOCAL_INPUT_VARIABLE(input_0,
    char, push_button, no_converter, NULL)
  LOCAL_OUTPUT_VARIABLE(output_0,
    char, led, no_converter, NULL))
```

Figure 4. Sample local I/O device configuration.

cyclic events generated by the timer task and consist of three phases. The *computation* phase (C phase) is implemented by the user-written control function. It operates on input PI variables collected in the previous cycle and computes output PI variables. It is followed by an *output* phase (O) that commits output PI variables and by an *input* phase (I), which collects input PI variables for the next cycle.

The same control function also handles sporadic events sent by UI tasks. Since the control task is single-threaded, two different kinds of interference are possible.

1) Any sporadic event generated while cyclic activities are in progress is postponed until the control task has completed them, as depicted in the left part of Figure 3.
2) Sporadic event handling (S) runs to completion after it starts. Therefore, as shown in the right part of the figure, if a cyclic event arrives while sporadic event handling is in progress, the start of the next cycle is delayed.

No provisions are made to bound the first kind of interference, since UI tasks are not considered hard real-time. On the other hand, the worst case for the second kind of interference is upper-bounded by the event system prioritization mechanism, which always delivers cyclic rather than sporadic events when both are available. As depicted in the right part of Figure 3, denoting with S_i the worst-case handling time for sporadic event i, the maximum interference is bounded by $\max_i(S_i)$, regardless of the number of pending sporadic events.

As an additional safeguard against dedicating excessive (high-priority) control task execution time to sporadic events, the event system implementation also prevents UI tasks from making more than a configurable number k of sporadic events pending. The implementation is based on a simple back pressure mechanism within the sporadic event generation primitive, better detailed in Section III-B.

D. Configuration Workflow

The last part of user-written code to be discussed in this paper and shown in the bottom-left corner of Figure 2, is the *I/O and shared memory configuration* files. The I/O configuration file contains a set of C macro invocations that define which local and remote I/O devices are configured in the system, as well as their relationship with PI variables. Quite intuitively, the shared memory configuration file plays a similar role concerning the structure of shared memories [B] and [C] and their relationship with PI variables, through the mirroring mechanism discussed in Section II-B.

As an example, Figure 4 shows how a simple local I/O device is configured. The device is called `instance_name` and belongs to device class `sample_class`. Class-related information is retrieved from a device database, not discussed here for conciseness. In this particular case, it is known from the database that devices belonging to `sample_class` implement a physical input `input_0` and a physical output `output_0`.

The I/O configuration file establishes that `input_0` is connected to a push button whose state will be accessible by means of PI variable `char push_button`, without any conversion. In fact, the configuration states to use the `no_converter` conversion function, with `NULL` parameter. The configuration contains similar information about the physical output, which is connected to a LED accessible through PI variable `char led`.

Using this approach, the control function works directly on `push_button` and `led`, using ordinary C-language statements, and the framework reflects their values to/from the corresponding physical I/O points in a completely transparent way. Thus, redesigning the system by moving an I/O point from a local device to a remote I/O board, for instance, becomes a matter of updating the I/O configuration file and does not require any changes to the control code. On the other hand, the firmware must be rebuilt, because configuration files are currently parsed at compile time and the framework does not support dynamic reconfiguration.

III. Implementation Highlights

This section complements Section II by highlighting two important implementation aspects and providing more details about them.

A. Input-Output Abstraction

One of the main framework design goals is the complete separation of the user-written control function from I/O-related details. Figure 5 outlines the multi-stage process that the framework implements to reach this goal. More specifically, the figure represents how the framework handles the `push_button` input PI variable, declared as shown in Figure 4. Output variables are handled in a similar way.

During the I phase of the control cycle (see Figure 3) the communication module scans the PI table, depicted at the bottom left of Figure 5. For each variable, the framework locates and invokes the appropriate device driver input method (which is responsible of retrieving the value of the variable from the device it resides on) and converter function (to convert the variable from its device-specific representation into a format suitable for the control function). This is done by following appropriate pointers rooted at the PI table.

As a result, in our example the current state of the push button is reflected into variable `char push_button`. The user-written control function can then use this variable during the C phase of the control cycle. No explicit synchronization mechanisms are needed because the I and C phases are executed sequentially within the control task. The framework

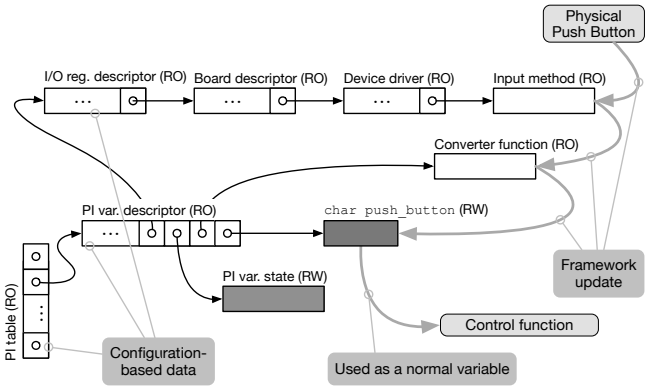

Figure 5. I/O abstraction process and related data structures.

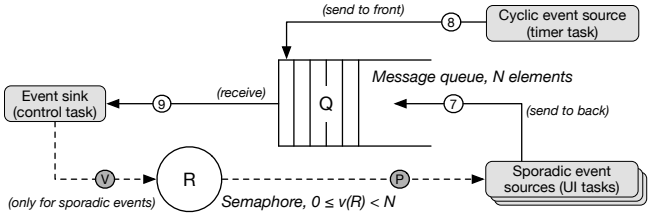

Figure 6. Detailed view of event system implementation.

also associates and optionally makes available to the user additional per-variable state information, which summarizes the outcome of the operations just mentioned.

It is important to remark that the framework defines all data structures shown in Figure 5 automatically, starting from configuration items like the one shown in Figure 4. With the exception of PI variables themselves (like `push_button`) and their state information, all these data structures are completely hidden from the user. Furthermore, by means of a careful data structure design and layout, it was possible to share information as much as possible and store most data in read-only Flash memory.

As an example, board descriptions and device driver data structures are known to be immutable and are shared among all boards belonging to the same class. This is convenient to minimize memory consumption and, even more, save read-write memory that is often scarce in low-cost embedded systems. Referring back to Figure 5, the only read-write structures—to be stored in RAM—are the dark-gray-colored ones.

B. Event System

The event system consists of a FREERTOS message queue Q and a counting semaphore R. Figure 6 portrays a more detailed view of its implementation. For consistency, labels on event flows have been kept the same as in Figure 2. Instead, labels on semaphore primitives are enclosed in gray circles and summarize the primitive itself (`P` corresponds to *take* and `V` corresponds to *give*, when using FREERTOS's nomenclature).

The main event system design goal is to support generation and buffering of up to N events, of which $N-1$ are sporadic and one is cyclic. As remarked in Section II-C, in order to bound control cycle jitter due to interference with sporadic event handling, the cyclic event must be given higher priority than the others. This goal is accomplished by means of two distinct mechanisms:

1) Although FREERTOS does not implement fully prioritized message queues, it does provide a primitive to send a message to the *front* of a queue rather than the back. The event system makes use of this feature when sending a cyclic event to Q. Sporadic events are sent to the back, and hence, handled in a FIFO fashion.
2) By itself, prioritizing operations on Q is insufficient to grant cyclic events expedited handling, if Q is completely full when a cyclic event occurs. In this case any send operation—regardless of whether it is directed to the front or to the back—blocks the caller and FREERTOS does not offer any built-in mechanism to address this issue. Therefore, R is used as a rate control semaphore to bound the number of pending sporadic events to $N-1$, and hence, reserve one element of Q for cyclic events at any time.

To implement rate control, R is used according to the following protocol:

- the initial value of R is $N-1$;
- before sending a sporadic event to Q, the event system performs a P on R, blocking if necessary;
- after receiving a sporadic event from Q, the event system performs a V on R.

As a side effect, neglecting transients in between semaphore and queue operations, the current value of R, denoted as $v(\mathrm{R})$ in Figure 6, is always $0 \leq v(\mathrm{R}) < N$ and represents how many elements of Q are available to sporadic events.

IV. EXPERIMENTAL EVALUATION

This part of the paper reports on the performance of the proposed framework and estimates its overhead. The experimental setup conforms to the architecture portrayed in Figure 1 and consists of a controller card, based upon the LPC2468 microcontroller [17], connected to a remote I/O card through a Modbus-CAN [16] bus. A local I/O device configured as shown in Figure 4 is also part of the setup. Footprint information has been derived from link-time information, while execution performance has been assessed by instrumenting the framework code with timestamping points.

The resolution of the standard FREERTOS time services is limited by the tick timer frequency, which is set to 1 kHz by default and cannot be increased significantly without incurring unacceptable overhead. For this reason, a separate 32-bit hardware counter was used for timestamping, as in [18]. It runs at the same clock speed as the CPU, that is, 72 MHz, thus reaching a resolution of about 14 ns.

Since counter registers are readily accessible to the CPU, timestamping is performed in less than 10 machine instructions, which include storing the timestamp into fast static RAM. The timestamping execution time S was estimated by taking two timestamps consecutively and calculating their difference. Over 2000 samples, the average execution time was found to be $\mu_S = 0.48\,\mu s$ with negligible variance $\sigma_S^2 < 2.13 \cdot 10^{-4}$.

This estimate might be marginally optimistic, since the compiler might be able to optimize timestamping operations

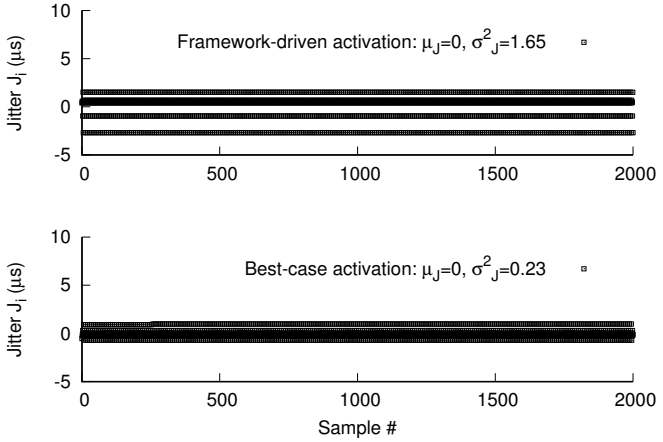

Figure 7. Experimental time traces for control function activation.

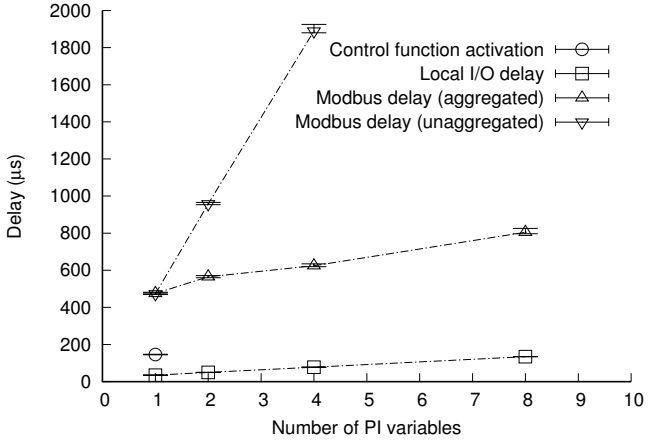

Figure 8. Delays introduced by main framework activities.

better than in other cases, if they are very close to each other. Nevertheless, it confirms that timestamping is not invasive when considering time intervals of tens of microseconds at least. The maximum time interval that can be measured without wraparounds is around 59 s and static RAM capacity is sufficient to hold up to about 4000 timestamps.

A. Execution Performance and Overhead

Execution performance evaluation revolved around the following two aspects, both related to framework overhead.

1) The ability of the framework to execute the C phase of the control cycle with an accurate *period* and acceptable *jitter*.
2) The *delay* introduced by the main framework activities, as a function of the number of PI variables and the kind of board being used.

For what concerns the first aspect, best-case performance is attained when the C phase is triggered directly by the underlying RTOS timing mechanism, `vTaskDelayUntil` for FREERTOS, without interposing any other software layer. Any difference between this term of reference and the actual framework-controlled C phase activation performance is due to the overhead of the framework itself—more specifically, its even system. Denoting by $c_0, c_1, \ldots$ the sequence of *nominal* C phase activation times, equally spaced by the nominal cycle period $P = c_i - c_{i-1} \ \forall i > 0$, the jitter J_i affecting the i-th C phase can be determined by measuring the sequence of *actual* C phase activation times $c'_0, c'_1, \ldots$ and calculating the actual cycle period P'_i of the i-th C phase as:

$$P'_i = c'_i - c'_{i-1} \ \forall i > 0. \tag{1}$$

Then, J_i is given by the difference between the actual and nominal period of the i-th C phase, that is:

$$J_i = P'_i - P \ \forall i > 0, \tag{2}$$

where positive values of J_i denote a late activation and negative values denote an early activation. The mean value of J_i over all i (denoted by μ_J) is expected to be zero and any non-zero value indicates a systematic error of the actual periods versus the nominal ones. The variance of J_i (denoted by σ_J^2) represents the jitter magnitude. Experimental results for best-case activation over 2000 samples are shown at the bottom of Figure 7. They manifest the absence of systematic timing errors ($\mu_J = 0$) and exhibit an activation jitter $\max_i |J_i| < 1\,\mu$s with $\sigma_J^2 = 0.23$. The top part of the same figure also shows the activation jitter with the interposition of the framework event system depicted in Figure 6 between the cyclic event source and the control task. The experimental results confirm that the jitter increases to a $\max_i |J_i| < 3\,\mu$s with $\sigma_J^2 = 1.65$, but it is still negligible with respect to the minimum cycle period the framework and the underlying RTOS support, that is, 1 ms. At the same time, observing that it is still $\mu_J = 0$ rules out any systematic timing errors the framework might introduce.

Regarding the second aspect under evaluation, Figure 8 depicts the two most important sources of overhead within the framework. Both have been evaluated experimentally and affect the minimum attainable cycle time. Namely:

1) the delay introduced by the event system shown in Figure 6 when activating the control function;
2) the total time needed by the O and I phases depicted in Figure 3.

The second delay has been measured as a function of the number of configured PI variables, also taking into account the kind of board they reside on (local versus remote) and the possibility of aggregating multiple variable updates into a single bus transaction (for remote Modbus boards only). On the other hand, the first delay has been measured only once (and is shown near the bottom left corner of the figure) because it is independent from all those factors. In each plot, the symbol is placed on the mean delay calculated over 2000 samples and whiskers extend to the minimum and maximum measured delay. Only input PI variables have been considered in the experiments, because outputs cause analogous delays.

The results presented in Figure 8 show that the control function activation delay is below $200\,\mu$s and is negligible with respect to other delay sources, especially when remote boards are involved. As expected, I/O delays are approximately linear with respect to the number of PI variables. For remote boards, experimental data also confirm that Modbus transactions dominate the delay. On the other hand, I/O delays for local boards provide a reasonable estimate of framework overhead in this area. This is because, in the experimental setup local I/O transactions introduce a negligible amount of delay by themselves and all the rest can be attributed to the

TABLE I. MEMORY FOOTPRINT DIVIDED BY CATEGORY

Module	*Text and RO data (B)*	*RW Data (B)*	*BSS (B)*
Framework	9938	0	424
Framework Utilities	5044	0	0
Main program	2159	0	228
Local I/O library	260	0	0
Modbus I/O library	26877	0	798
RTOS	16154	8	33052
C runtime library modules	30092	1284	94482
Total	90524	1292	128984

I/O abstraction performed by the framework (see Figure 5). It should also be noted that aggregation brings a significant performance improvement because it trades extra Modbus transactions for a more complex local data processing, at a fraction of the cost. Last, but not least, data are in good agreement with respect to [16]. Any marginal performance improvement in the present case can easily be justified by considering that a faster Modbus slave has been used.

B. Memory Footprint

The same firmware considered in Section IV-A was also evaluated to determine another kind of overhead, that is, its memory footprint. The evaluation was performed statically (on the firmware object code), by means of the `size` toolchain command, leading to the results listed in Table I. The rows of the table correspond to different parts of the firmware, while columns divide footprint into three standard categories: text and read-only (RO) data, read-write (RW) initialized data, and read-write uninitialized data (BSS). These categories are important from the practical point of view because, depending on the target system, they may correspond to different kinds of memory. For instance, text and RO data can conveniently be stored in Flash memory (if available on the target) rather than RAM.

The memory footprint of framework components is given in the first three row of the table. More specifically, we considered:

- the real-time part of the *framework* itself, described in Sections II and III, which also allocates storage for PI variables, as well as their descriptors and states;
- additional *framework utilities*, consisting of functions to dump framework data structures in human-readable format and other debugging aids;
- the *main program*, which allocates the main configuration and board state data structures, besides implementing an exemplar control function.

The rows that follow pertain to other firmware components that are used by the framework, but would be required even without it, namely, the I/O libraries, operating system and C runtime library modules. The results shown in Table I confirm that the framework footprint is acceptable and does not significantly impact overall memory requirements with respect to other major components. Indeed, the total footprint of core components is only 17141 B + 652 B (Text and RO data + BSS), which is lower than the footprint of the Modbus I/O library alone, for instance.

For the sake of completeness, it is worth to remark that the RTOS BSS footprint includes the memory pool from which memory for *task stacks* and all other objects managed by the operating systems is drawn (32 KB). Similarly, the C runtime library BSS includes the library *heap* (64 KB), which satisfies all dynamic memory allocation requests made by the application code and the library itself, as well as the *other stacks* required by the processor, except task stacks (28 KB).

V. CONCLUSION

This paper presented the design of a firmware development framework whose aim is to speed up application development and deployment with respect to plain C-language programming. The framework has been implemented and its performance experimentally evaluated with satisfactory results, regarding both execution time overhead and memory footprint.

Foreseen future work includes testing the framework in the context of a real-world embedded application, as well as further extending its remote communication capabilities to Ethernet-based networks and protocols.

REFERENCES

[1] M. Samek, Practical UML Statecharts in C/C++, 2nd ed. Newnes, Oct. 2008.

[2] W. Bolton, Programmable Logic Controllers, 6th ed. Newnes, Mar. 2015.

[3] CODESYS, Industrial IEC 61131-3 PLC programming, available online, at https://www.codesys.com [retrieved: Mar. 2017].

[4] T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sunder, A. Valentini, and A. Martel, "Framework for distributed industrial automation and control (4DIAC)," in Proc. 6th IEEE International Conference on Industrial Informatics (INDIN), Jul. 2008, pp. 283–288.

[5] IEC 61131-3, Programmable controllers — Part 3: Programming languages, 3rd ed., International Electrotechnical Commission, Feb. 2013.

[6] Arduino AG, Arduino IDE, available online, at https://www.arduino.cc [retrieved: Mar., 2017].

[7] P. Buonocunto, A. Biondi, and P. Lorefice, "Real-time multitasking in Arduino," in Proc. 9th IEEE International Symposium on Industrial Embedded Systems (SIES), June 2014, pp. 1–4.

[8] ISO/IEC 9899, Programming Languages — C, 3rd ed., International Organization for Standardization and International Electrotechnical Commission, Dec. 2011.

[9] R. Barry, Using the FreeRTOS Real Time Kernel – Standard Edition, 1st ed. Raleigh, North Carolina: Lulu Press, 2010.

[10] ChaN, FatFs Generic FAT File System Module, available online, at http://elm-chan.org/fsw/ff/00index_e.html [retrieved: Mar. 2017].

[11] A. Dunkels, lwIP – A Lightweight TCP/IP stack, available online, at http://savannah.nongnu.org/projects/lwip/ [retrieved: Mar. 2017].

[12] I. Cibrario Bertolotti and T. Hu, "Modular design of an open-source, networked embedded system," Computer Standards & Interfaces, vol. 37, Jan. 2015, pp. 41–52.

[13] C. D. Locke, "Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives," Real-Time Systems, vol. 4, no. 1, 1992, pp. 37–53.

[14] M. Caccamo, T. Baker, A. Burns, G. Buttazzo, and L. Sha, "Real-time scheduling for embedded systems," in Handbook of Networked and Embedded Control Systems, D. Hristu-Varsakelis and W. S. Levine, Eds. Birkhäuser Boston, 2005, pp. 173–195.

[15] T. P. Baker and A. Shaw, "The cyclic executive model and Ada," in Proc. IEEE Real-Time Systems Symposium (RTSS), Dec. 1988, pp. 120–129.

[16] G. Cena, I. Cibrario Bertolotti, T. Hu, and A. Valenzano, "Design, verification, and performance of a MODBUS-CAN adaptation layer," in Proc. 10th IEEE International Workshop on Factory Communication Systems (WFCS), May 2014, pp. 1–10.

[17] LPC2468 Product data sheet, rev. 4, NXP B.V., Oct. 2008, available online, at http://www.nxp.com/ [retrieved: Mar. 2017].

[18] I. Cibrario Bertolotti and T. Hu, "Real-time performance of an open-source protocol stack for low-cost, embedded systems," in Proc. 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Sep. 2011, pp. 1–8.